Behavioural Model Based Testing of Software Product Lines: Research Abstract

Devroey, Xavier

Published in:
Proceedings of the 18th International Software Product Line Conference - Volume 1

Publication date:
2014

Document Version
Peer reviewed version

Behavioural Model Based Testing of Software Product Lines

Research Abstract

Xavier Devroey
PReCISE Research Center, Faculty of Computer Science,
University of Namur, Belgium
http://directory.unamur.be/staff/xdevroey/
xavier.devroey@unamur.be

ABSTRACT

Since the inception of Software Product Line (SPL) engineering, concerns about testing SPLs emerged. The large number of possible products that may be derived from a SPL induces an even larger set of test-cases, which make SPL testing a very challenging activity. Some individual solutions have been proposed, but few are integrated in a complete testing process. In this paper, we summarize our research addressing variability-aware behavioural model-based testing. So far we developed a statistical prioritization technique and we have defined behavioural coverage criteria dedicated to behavioural model of a SPL. Our overall goal is to form an end-to-end model-driven approach, relying on Featured Transition System (FTS), a compact formalism to represent the behaviour of a SPL, where test-cases selection techniques are automated and made practical to the engineers. The formality of the envisioned models also makes them amenable to model-checking, yielding innovative combinations of quality assurance activities. The evaluation will be performed in two phases: using standard approaches (fault seeding and mutation testing); using the approach on an industrial case-study.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Reliability, Verification

Keywords

Software Product Line, Model-Based Testing, Featured Transition System

Doc. Symp., SPLC'14 , September 15 - 19, Florence, Italy

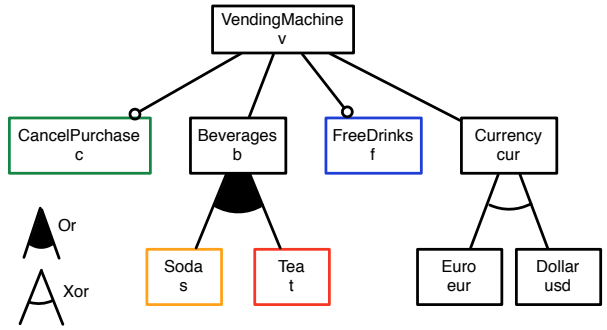
1. INTRODUCTION

By analogy with the manufacturing industry, the Software Product Line (SPL) [32] engineering paradigm relies upon the idea that families of software systems can be built by systematically reusing assets. Some are common to all family members, and others are only shared by a subset of the family [29]. Commonalities and variabilities amongst members of a SPL are represented by means of a Feature Diagram (FD) [20], while the individual specifications and design of these assets may be modelled using languages such as UML. For example, Fig. 1a presents the FD of a soda vending machine [7]. A common semantics associated to a FD d, noted $[\![d]\!]$, is the set of all the valid products allowed by d.

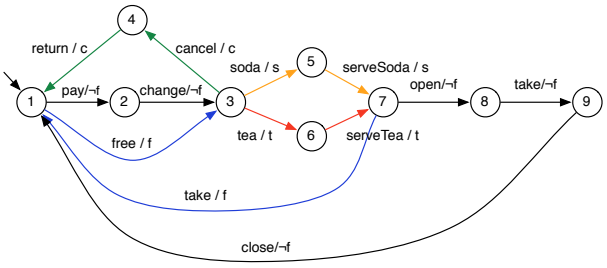
As for any software engineering paradigm, appropriate Quality Assurance (QA) techniques must be devised to increase confidence into the products. Research in this area comprise two kinds of approaches: *model checking* and *testing*. Model Checking aims at checking that a given property holds for a given model [2, 6, 23]. Testing aims at assessing that programs and other artefacts behave as expected [25], i.e., as described by the model. The notion of expected behaviour may cover different aspects: the expected output is given by the program (Functional Testing); the time it takes to the program to compute this output (Performance Testing); the ease of use of its graphical interfaces (UI Testing); etc.

In SPL engineering, the definition of such techniques is very challenging due to the combinatorial explosion induced by SPL variability. E.g., testing a SPL [26] means defining test-cases for every product. Given that for each FD with N features there are (at most) 2^N possible products, exhaustive testing of SPL is not practical for realistic SPLs. This research topic, despite being identified since 2001 [26], is still immature [15]. Strategies to test some selected products exist, but the selection of representative products is still in its initial stage [14].

We believe that SPL testing research may benefit from advances in related SPL research communities: the significant progress in the model checking community to combat combinatorial explosion and devise efficient checking techniques [7]; and the adaptation of combinatorial interaction testing techniques to the SPL domain [30]. In non-SPL software engineering, Model Based Testing (MBT) [37] is a well-established discipline in which test-cases are generated from model(s) representing the specification of a system under



(a) Feature Diagram (FD)



(b) Featured Transition System (FTS)

Figure 1: The soda vending machine example [6]

test. These promising results motivate our will to apply MBT techniques to perform practical testing of SPLs. In this thesis, we will focus on the usage of a formal behavioural model (based on Transition Systems) of product lines used to generate test-cases according to given test criteria.

The rest of this paper is decomposed as follow: section 2 presents the context of this research and the background used in the rest of this paper. Section 4 presents the methodology we will follow to answer the research questions presented in section 3. Section 5 presents our contribution so far with key references in sections 6 and the thesis structure in section 7. Section 8 presents related works and finally, section 9 presents planned future works and our research agenda.

2. CONTEXT

Testing issues in SPL developments are reported for years [26] but still receive little attention [15]. A mapping study realised by do Carmo Machado et al. in 2014 [14], presents an overview of testing in SPL engineering. This study points out that the selection of representative products is still in its initial stage. One solution to select representative products is to use a feature coverage criteria to ensure that each feature, couple or t-uple of features is present in at least one tested product. For instance, in pairwise testing, the test-cases will correspond to the minimal set of products in which all pairs of features occur at least once. Pairwise testing has been generalized to T-wise testing but at the cost of a scalability problem. Perrouin et al. [31] describe a solution to split T-wise combinations into solvable subsets using Alloy. Oster et al. [28] describe a methodology to apply combinatorial testing to a feature model. The two approaches have been compared [30].

The verification community has made significant progress to combat such explosion and devise efficient verification techniques. These developments have been made at the domain engineering level (i.e., applicable for the whole SPL) by finding compact models and transformations to use state of the art solving technology. We present in the next subsection Featured Transition System (FTS) [7], a compact formalism to represent the behaviour of a complete product line.

2.1 SPL Behavioural Model

Transition Systems are used to model the behaviour of a system [2]. To allow the explicit mapping from feature to SPL behaviour, Featured Transition Systems (FTS) were proposed [7]. FTS are transition systems (TS) where each transition is labelled with a feature expression (i.e., a boolean expression over features of the SPL), specifying which products can execute the transition. Formally, a FTS is a tuple $(S, Act, trans, i, d, \gamma)$ where:

- S is a set of states;
- Act a set of actions;
- $trans \subseteq S \times Act \times S$ is the transition relation (with $(s_1, \alpha, s_2) \in trans$ sometimes noted $s_1 \xrightarrow{\alpha} s_2$);
- $i \in S$ is the initial state;
- d is a FD;
- $\gamma : trans \to [\![d]\!] \to \{\top, \bot\}$ is a total function labelling each transition with a boolean expression over the features, which specifies the products that can execute the transition.

For instance: $\neg f$ in Fig. 1b indicates that only products that do not have the $free$ feature may fire the $pay, change, open, take$ and $close$ transitions. A Transition System (TS) modelling the behaviour of a given product is obtained by removing the transition whose feature expression is not satisfied by the combination of features describing the product.

One possibility would be to use FTSs to directly model behavioural product lines. Unfortunately, FTS are very efficient for verification and analysis but are not meant to be used by engineers. Cordy et al. defined *Featured State Machines* (FSTM) [9] a formalism based on Harel's Statecharts that has already been used in industry. FSTMs may be transformed to FTSs that will be processed by the tools.

ProVeLines is a dedicated model checker for FTSs [9]. It includes the following functionalities that will be re-used for testing purposes:

- The Reachability, simulation, and Linear Temporal Logic (LTL) properties verification are used for test-case generation and test-case execution by combining model checking and testing techniques.
- The FSTM input language is more abstract than FTS and accessible to software engineers to model the behaviour of a SPL.
- The Textual Variability Language (TVL) supports a rich FD description language.
- The extensible architecture allows to add plugins.

ProVelines is not a layer above an existing model checker, it has been designed and implemented from scratch specifically to model check SPL behaviours. To the best of our knowledge, ProVeLines is the only model checker that verify properties for *all* the products of the SPL. In case of property violation, it gives *all* the products that violate the property.

3. RESEARCH QUESTIONS

In this thesis, we assume that **Model-Based Testing** (MBT) techniques can be used to test SPLs ($Hyp.1$). In particular, we focus on behavioural product lines, which can be formalized by means of **FTSs** and higher level languages.

In order to perform MBT, test engineers define **SPL testing criteria**. To this end, an appropriate test selection language has to be defined. We hypothesize that such criteria will involve both **variability and behavioral concerns** ($Hyp.2$). For instance, *pairwise coverage* can be combined with structural coverage of state machines (states, transitions). Additionally, existing **test-cases generation** methods for behavioural models have to be adapted to the SPL paradigm. To do so, **advances made by the SPL model checking community [5]** are assumed to be beneficial for test-case generation ($Hyp.3$).

From $Hyp.1$ and $Hyp.3$, we derive the following research question (**RQ1**): *How to benefit from SPL model checking techniques automation possibilities while still being accessible to test engineers with no specific knowledge in model checking?* E.g., adapt ProVeLine symbolic algorithm to generate abstract test-cases from a FTS.

From $Hyp.2$, we derive the following research question (**RQ2**): *How to define and select test-cases for a SPL based on this FTS model?* The test-cases we will generate will be test-cases for a SPL (and not a product) and will contain variability information. Similarly, test-case selection criteria will operate both at behavioural and variability level.

From $Hyp.1$, we derive the following research question (**RQ3**): *How to combine and integrate selection and prioritization techniques on FTSs in an end-to-end traceable framework?* FTS is an abstract formalism, not suited for software engineers, and lacks hierarchical constructions.

4. RESEARCH METHODOLOGY

This thesis will make the point that involvement of the verification and testing communities is necessary to advance the SPL QA state of the art. We work at the theoretical level to define novel SPL test selection and generation techniques in relationship with the verification community. But SPL QA is also a practical challenge for engineers. We also work iteratively with MBT users and SPL engineers amongst our industrial partners to ensure the relevance and validate our research results. This research approach ensures that contributions are scientifically grounded but also applicable in industry (our main thread to validity). The main steps during the development of this thesis are described hereafter and will be executed in an iterative and incremental fashion as described in figure 2.

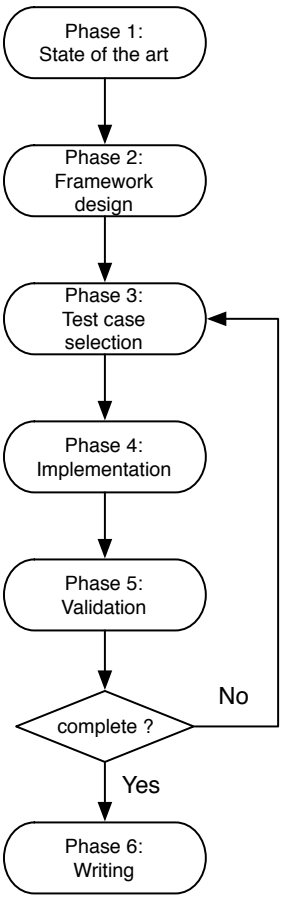


Figure 2: Research Methodology

Phase 1: State of the art. The state of the art covers three elements: Model-Based Testing [25, 37], SPL Testing [15, 14] and SPL Behavioural Model Checking literature [2, 5, 7]. We have also initiated a literature monitoring process that will continue until the end of the thesis.

Phase 2: Framework design. The second phase is concerned with the general design of a framework able to answer the research questions defined in section 3 [10].

Phase 3: Test cases selection and prioritization. In the third phase we define test-case selection strategies. So far, those strategies include statistical test-case prioritization [11] and coverage criteria [12].

Phase 4: Implementation. Techniques and methods defined during step 3 will be implemented in Java using Maven in order to decompose the different elements in components that may be rearranged. The current and future implementations may be found at `https://staff.info.unamur.be/xde/fts-testing/`.

Phase 5: Validation. In order to answer **RQ1**, we need to validate the techniques and methods and their implementation from a theoretical point of view, but also with test engineers. For each algorithm, a first validation in done internally to study the correction, completeness and complexity of the algorithm. A classical approach in software testing is to use mutation testing to assess a test-case generation algorithm [1, 16,

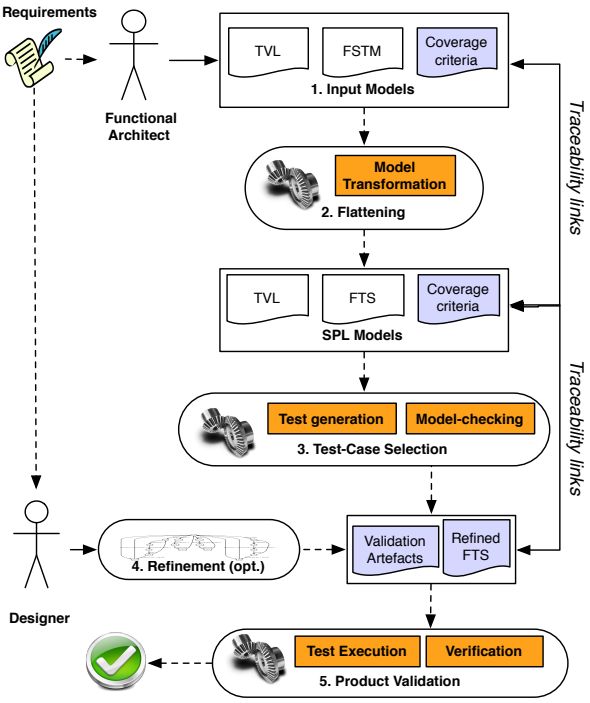


Figure 3: Approach Overview

18, 19]. We plan to develop mutation testing for FTSs by adapting classical TS operators and add variability-aware operators. The second validation assess the relevance of the approach against an industrial case study using an empirical approach [40]. This industrial case study may be from one of our industrial partners or an open source system.

Phase 6: Writing. Once the framework designed in phase 2 is implemented and validate, we will write the thesis.

5. CONTRIBUTIONS

This thesis intends to answer the research questions by offering a framework for practical SPL testing and model-checking. This framework relies on FTS, supports automated test-case generation and verification techniques, is made practical to the engineers by means of transformation from software engineer-oriented modelling language (FSTM). We performed a preliminary study and review of the SPL testing and behavioural MBT literature in order to review the state of the art and to sketch a first vision of the approach [10].

Figure 3 presents an overview of the approach [10]. The first step of the process consists in designing the input models. We use FSTM [9] as model of the behaviour of the product line. The FD is designed using the Textual Variability Language `TVL` (see Chapter 8 of [5] for details). Coverage criteria are defined for the SPL on FTSM elements. We described the test selection problem in FTSs as a trade-off between the coverage, the number of test-cases, and the number of products to test [12]. We considered the classical criteria: *state-coverage*, *transition-coverage*, *transition-pair-coverage*, and *path-coverage*. In our future work, we intend to extend those criteria to SPL coverages: e.g., the number of products covered by a test-suite, the number of features, the number of pairs of features, etc.

Model transformations will translate the models and test criteria from step 1 into processing-oriented artefacts, namely FTS and `TVL` as well as translated criteria. We plan to take advantage of the various model transformation environments available today (e.g., ATL, Kermeta) to actually implement these transformations.

The third step of our process is concerned with test-cases selection using model checking and test generation techniques. Generated abstract test-cases may be refined in step 4 by the designer who has a deeper knowledge of the product line (e.g., by refining actions in a sequence of transitions), giving a refined FTS and a validation model (i.e., more detailed abstract test-cases). We presented a first test generation technique by combining statistical testing techniques with FTSs in order to prioritize product testing [11]. The result of the process is a refined FTS representing a subset of the original FTS that has to be assessed (using testing and/or model checking) in priority.

In the last step, the refined FTS is used to perform model checking while the test-cases (part of the validation artefacts) are concretized in order to be executed on the products using existing techniques [37]. The last step is to present the test results in a meaningful way to the test engineer. Indeed, since many transformations have been performed automatically, it may be difficult to understand the raw results. Our goal is to complete the validation chain by illustrating violations and/or failures in the terms of the original models (e.g. providing pruned visualizations of feature diagrams showing the features that have been involved in a failed test or highlighting transitions in the FSTM model) thanks to traceability links.

Although other steps are important, some presents interesting research challenges (e.g., traceability and evolution issues as stated by do Carmo Machado et al. [14]), some may be solved using standard techniques. This thesis will focuse on step 3: Test Generation. Test generation is a broad topic, we work by considering standard testing techniques and try to transpose them to the SPL/FTS paradigm. So far, we worked on *test-case selection* and *test-case prioritization*. In the near future we intend to explore mutation techniques for FTSs in order to assess our test-case selection algorithms using a standard approach [25].

5.1 Test-Case Selection

In our previous work [12], we present the notions of abstract test-case and the abstract test-case generation problem according to classical TS selection criteria.

Abstract Test-Case and Executable Abstract Test-Case. Basically, an abstract test-case is a sequence of actions $(\alpha_1, ..., \alpha_n)$ in a FTS such as there exists a continuous sequence of transitions labelled with those actions $(i \xrightarrow{\alpha_1} s_k \xrightarrow{\alpha_2} ... \xrightarrow{\alpha_n} s_n)$. An executable abstract test-case is a sequence of actions such as there exists a product which may fire a continuous sequence of transitions labelled with those actions. For instance, on the FTS in figure 1b, the sequences $\{pay, change, cancel, return\}$ and $\{pay, change, soda, serveSoda, take\}$ are both valid abstract test-cases (there exists 2 sequences of transitions with those actions), but only the first one may be fired by a valid product of the product line (product with $\neg f$ and c features selected). The

second abstract test-case mixes free (f) and non-free ($\neg f$) machines behaviours.

Coverage Criteria. We define a coverage criteria as the percentage of coverage for a given set of abstract test-cases over a particular FTS. Coverage criteria are defined as functions returning for a FTS and a set of abstract test-cases a real value between 0 and 1. For instance, the $all - states$ coverage criteria gives the ratio between the number of states reached when executing the test-cases and the number of states in the FTS. Other considered criteria are: *all-transitions*, *all-transition-pairs*, and *all-paths*.

Executable Abstract Test-Case Selection. Using those notions, we define the executable abstract test-case selection problem for a given selection criteria as a trade-off between three values:

- minimizing the number of executable abstract test-cases;
- maximizing the coverage targeted for the selection criteria:
- minimizing the number of products needed to execute all the test-cases.

We believe that this trade-off depends on the system under test and the goal of the test engineer.

First Validation Using Fault Seeding. We devised an FTS-aware random test generation strategy (e.g. systematically producing random executable abstract test-cases) and compare randomly generated test-cases with selection criteria [12, 13]. The random test generation consist in a random walk in the FTS such that there exists a product able to execute this walk. To compare the random test-cases and coverage criteria-driven generated test-cases, we simulate bugs in the system by seeding faults in the FTS. So far a fault is a state, a transition or an action (resp.) which trigger the bug when it is traversed, fired or executed (resp.). The number of faults detected by a test-case corresponds to the number of faulty states, transitions and actions traversed, fired or executed when executing the test-case. A first comparison between random test-cases and *all-states* test-cases has been presented [13]. Another widespread validation technique to assess the quality of a coverage criteria is the computation of a mutation score, corresponding to the number of mutants killed by a (generated) test-set [25]. This will be explored in the next months.

We also plan to combine such criteria with each other and with test-case selection based on temporal properties. Our approach is implemented using the ProVeLines family of SPL model-checkers [8].

5.2 Test-Case Prioritization

In our previous work [11], we developed a statistical test-case prioritization technique based on the behaviour of running systems (i.e., products). Statistical prioritization is particularly useful during regression testing to ensure as quickly as possible that the most commonly used functionalities of a system still work as expected after an update.

Usage Model. Fig. 4 presents the overall process. First, a usage model (basically a labelled Deterministic Timed Markov Chain) is built using logs of the running systems (products of the product line) [17, 35]. E.g., for the soda vending machine presented in Fig. 1b, we have the usage

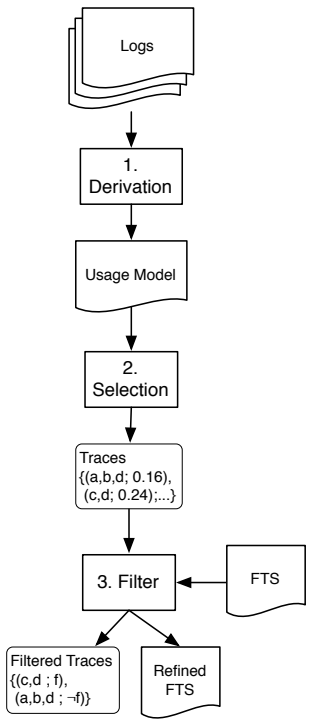


Figure 4: Prioritization overview [11]

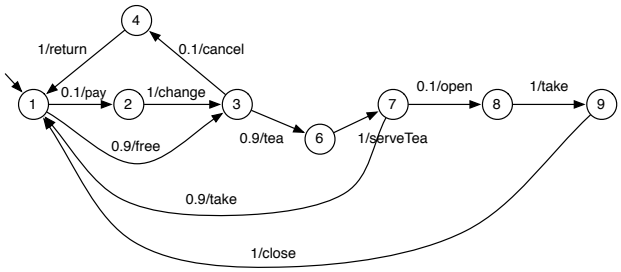
model in Fig. 5a. The usage model is built from execution traces of the system, some transitions may be missing (*soda* and *serveSoda* in our case) if the actions are never performed by the running system.

Trace Selection. Traces (i.e., sequences of actions) are selected in this usage model such as the probability of the trace to be executed is between a lower and upper bound given by the engineer. E.g., 0 and 0.1 in Fig. 5, which produces the following traces:

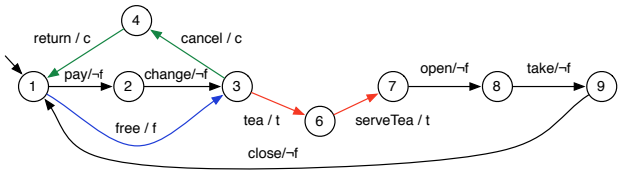
$$Select(0; 0.1; usagemodel_{svm}) = \{$$
$$(pay, change, cancel, return; 0.001);$$
$$(free, cancel, return; 0.09);$$
$$(pay, change, tea, serveTea, open, take, close; 0.009);$$
$$(pay, change, tea, serveTea, take; 0.09);$$
$$(free, tea, serveTea, open, take, close; 0.081)\}$$

Trace Filtering. The selected traces are filtered using the FTS to keep only executable traces (sequences of actions such as there exists at least one product able to fire transitions with those actions). Our example contains two unexecutable traces (i.e., not executable by at least one valid product of the product line in Fig. 1): (pay, $change$, tea, $serveTea$, $take$) and ($free$, tea, $serveTea$, $open$, $take$, $close$).

Product Prioritization. The output of the filter process is a set of executable traces ordered according to their probability to be executed by the system and a refined FTS representing the subset of the FTS to test and/or model check in priority. Each executable trace is coupled with the products that can execute it, giving a prioritized list of products to test. The refined FTS for the soda vend-



(a) Usage Model



(b) Refined Featured Transition System

Figure 5: The soda vending machine prioritization example [11]

ing machine in presented in figure 5b. If we take the trace $t = (pay, change, tea, serveTea, open, take, close)$, the products will have to satisfy the $\neg f \wedge t$ feature expression. This gives us a set of 8 products (amongst 32 possible):

$$\{(v, b, cur, t, eur); (v, b, cur, t, usd); (v, b, cur, t, c, eur);$$
$$(v, b, cur, t, c, usd); (v, b, cur, t, s, eur); (v, b, cur, t, s, usd);$$
$$(v, b, cur, t, s, c, eur); (v, b, cur, t, s, c, usd)\}$$

All of them executing t with a probability of 0.009 which is the least probable behaviour of the soda vending machine.

Feasibility Assessment. To assess the feasibility of the approach, we used a 5.26 Go Apache log of a running instance of Claroline, an on-line course management system[1] to derive a usage model [11]. We build the FD (44 features) manually and used a web crawler to build the FTS (107 states and 11236 transitions) partially automatically. We applied our technique and drastically reduced the size of the refined FTS: 69 states and 844 transitions with the widest interval($[10^{-7}; 1]$) of our experiment. Unfortunately, the FD contained too much variability to significantly reduce the number of products to test and other selection criteria had to be combined with this technique to have a proper product prioritization.

Future Work. Future works include combining the coverage criteria presented in section 5.1 and the refined FTS generation in order to get a finer grained list of prioritized products. We also plan to apply this technique to an industrial case of one of our partners. We intend to submit an extended version of [11] to a journal, with the complete validation and a connection to the work of Samih et al. [33].

6. KEY REFERENCES

Main publications (so far) are:

[10] "*A vision for behavioural model-driven validation of software product lines*" presented at ISoLA '12

[11] "'*Towards statistical prioritization for software product lines testing*" presented at VaMoS '14

[12] "*Coverage Criteria for Behavioural Testing of Software Product Lines*" accepted at ISoLA '14

[13] "*Abstract Test Case Generation for Behavioural Testing of Software Product Lines*" accepted at SPLat '14 (SPLC Workshop)

[1] `http://www.claroline.com/`

7. THESIS STRUCTURE

Hereafter are presented the thesis structure and percentage of completion upon submitting this paper:

1. **Introduction** (60%)
2. **Background** (80%)
   (a) Model-Based Testing
   (b) Software Product Line Engineering
      i. Software Product Line Testing
      ii. Behavioural Model Checking of Software Product Line
3. **Framework Overview** (50%)
4. **Algorithms and Implementation** (20%)
5. **Validation** (5%)
   (a) Mutation Testing
   (b) Users experience returns
6. **Results Discussion** (0%)
7. **Conclusion and Future Works** (0%)

8. RELATED WORK

Other strategies to perform SPLs testing at the domain level have been proposed, e.g., incremental testing in the SPL context [38, 28, 24]. Lochau et al. [24] defined a model-based approach that shifts from one product to another by applying deltas to state-machine models. These deltas enable automatic reuse/adaptation of test model and derivation of retest obligations. Oster et al. [28] extend combinatorial interaction testing with the possibility to specify a predefined set of products in the configuration suite to be tested. There are also approaches focused on the SPL implementation by building variability-aware interpreters for various languages [22]. Based on symbolic execution techniques, such interpreters are able to run one test-case on a very large set of products at the same time [21]. In [4], Cichos et al. use the notions of 150% test model, i.e., a test model of the behaviour of a product line, and of test goal to derive test-cases for a product line but do not define coverage criteria at the SPL level. In [3], Beohar et al. adapt the *ioco* framework [36] to FTSs. Contrary to this approach, we do not seek exhaustive testing of an implementation but rather to select relevant abstract test-cases based on the criteria provided by the test engineer.

9. RESEARCH AGENDA

In the nearest future, we intend to implement and extend coverage criteria [12] in order to *(1)* take into account classical transitions based coverage criteria, e.g., state coverage, transition coverage, transition pair coverage, path coverage, etc. *(2)* Use existing statistical testing tools (like MaTeLo) in order to connect to our statistical prioritization technique [11]. *(3)* validate the complete approach by concretizing the test-cases on two real systems: one coming from the open source community (like Claroline [11], the Drupal framework [34], or Wordpress), and one industrial systems developed by our partners (Thales, etc.) as recommended by Metzger et al. [27]. Our work/publication plan for the next months is the following:

Test case prioritization. We will extend and connect our work to Samih et al. [11, 33] using MaTeLo[2], a Model-Based Testing tool that use usage model to generate test-cases (without variability). We will submit our work to a journal (fall 2014).

Test case selection. The generation of test-cases according to the all-states coverage criteria is implemented. It has been compared with random test-case generation using a fault seeding algorithm (i.e., state, transitions and actions randomly selected and considered to be faulty) [13]. We plan to validate this algorithm using mutation testing techniques and to extend our work to other coverage criteria.

Mutation testing. Specific operators for mutation of FTSs will be developed and implemented in the next weeks. Those operators will be used to assess the test-case generation algorithms using coverage criteria. The complete implementation and the comparison of different coverage criteria using mutation scores will be described in a conference paper (winter 2014).

Next year (2015). We plan to integrate and apply the test-case generation and mutation testing in a coherent framework to perform Behavioural SPL Model-Based Testing. The transformation from FSTM and coverage criteria to FTS will be implemented and a traceability policy will be chosen [39]. Coverage criteria will be composed and refine to allow finer grained test-case selection (e.g., a specific set of states and/or transitions and/or features, etc.).

10. REFERENCES

[1] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *TSE*, 32(8):608–624, 2006.

[2] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[3] H. Beohar and M. R. Mousavi. Spinal Test Suites for Software Product Lines. *ArXiv e-prints*, 2014.

[4] H. Cichos, S. Oster, M. Lochau, and A. Schürr. Model-based Coverage-driven Test Suite Generation for Software Product Lines. In *MODELS'11*, pages 425–439. Springer-Verlag, 2011.

[5] A. Classen. *Modelling and Model Checking Variability-Intensive Systems*. PhD thesis, PReCISE Research Center, Faculty of Computer Science, University of Namur (FUNDP), 2011.

[2]see: `http://all4tec.net/index.php/en/model-based-testing/20-markov-test-logic-matelo`

[6] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of tvl. *Science of Computer Programming*, 76:1130–1143, 2011.

[7] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. J.-F. Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *TSE*, 39(8):1069–1089, Aug. 2013.

[8] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. ProVeLines: A Product Line of Verifiers for Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference Co-located Workshops*, SPLC '13 Workshops, pages 141–146. ACM, 2013.

[9] M. Cordy, M. Willemart, B. Dawagne, P. Heymans, and P.-y. Schobbens. An Extensible Platform for Product-Line Behavioural Analysis. In *SPLat@SPLC'14*, Florence, Italy, 2014. ACM.

[10] X. Devroey, M. Cordy, G. Perrouin, E.-Y. Kang, P.-Y. Schobbens, P. Heymans, A. Legay, and B. Baudry. A vision for behavioural model-driven validation of software product lines. In T. Margaria and B. Steffen, editors, *ISoLa'12*, volume 7609 of *LNCS*, pages 208–222. Springer, 2012.

[11] X. Devroey, G. Perrouin, M. Cordy, P.-Y. Schobbens, A. Legay, and P. Heymans. Towards statistical prioritization for software product lines testing. In *VaMoS'14*, pages 10:1–10:7, New York, NY, USA, 2013. ACM.

[12] X. Devroey, G. Perrouin, A. Legay, M. Cordy, P.-y. Schobbens, and P. Heymans. Coverage Criteria for Behavioural Testing of Software Product Lines. In T. Margaria and B. Steffen, editors, *ISoLa'14 (to appear)*, LNCS. Springer, 2014.

[13] X. Devroey, G. Perrouin, and P.-y. Schobbens. Abstract Test Case Generation for Behavioural Testing of Software Product Lines. In *SPLat@SPLC'14*, Florence, Italy, 2014. ACM.

[14] I. do Carmo Machado, J. D. McGregor, Y. a. C. Cavalcanti, and E. S. de Almeida. On strategies for testing software product lines: A systematic literature review. *Information and Software Technology*, 56(10):1183–1199, Oct. 2014.

[15] E. Engström and P. Runeson. Software product line testing-a systematic mapping study. *Information and Software Technology*, 2010.

[16] S. Fabbri, J. C. Maldonado, and M. E. Delamaro. Proteum/FSM: a tool to support finite state machine validation based on mutation testing. In *SCCC '99*, pages 96–104, 1999.

[17] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining Behavior Models from User-Intensive Web Applications Categories and Subject Descriptors. In *ICSE'14*, Hyderabad, India, 2014. ACM.

[18] C. Henard, M. Papadakis, G. Perrouin, J. Klein, Y. Le Traon, and Y. L. Traon. Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing. In *ICSTW'13*, pages 188–197, 2013.

[19] Y. Jia and M. Harman. An Analysis and Survey of the

Development of Mutation Testing. *TSE*, 37(5):649–678, Sept. 2011.

[20] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. Spencer Peterson. Feature-Oriented domain analysis (FODA) feasibility study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.

[21] C. Kästner, H. V. Nguyen, and T. N. Nguyen. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *ICSE'14*, Hyderabad, India, 2014. ACM.

[22] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward Variability-aware Testing. In *PFOSD '12*, pages 1–8, New York, NY, USA, 2012. ACM.

[23] K. Lauenroth, A. Metzger, and K. Pohl. Quality Assurance in the Presence of Variability. In S. Nurcan, C. Salinesi, C. Souveyet, and J. Ralyté, editors, *Intentional Perspectives on Information Systems Engineering*, pages 319–333. Springer Berlin Heidelberg, 2010.

[24] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity. Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In A. Brucker and J. Julliand, editors, *Tests and Proofs*, volume 7305 of *LNCS*, pages 67–82. Springer Berlin Heidelberg, 2012.

[25] A. Mathur. *Foundations of software testing*. Pearson Education, 2008.

[26] J. D. McGregor. Testing a software product line. Technical report, Carnegie-Mellon University, Software Engineering Institute, 2001.

[27] A. Metzger and K. Pohl. Software Product Line Engineering and Variability Management: Achievements and Challenges. In *FOSE'14*, pages 70–84. ACM, 2014.

[28] S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. *Software Product Lines: Going Beyond*, pages 196–210, 2011.

[29] D. L. Parnas. On the Design and Development of Program Families. *TSE*, SE-2(1):1–9, 1976.

[30] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. le Traon. Pairwise testing for software product lines: Comparison of two approaches. *Software Quality Journal*, 2011.

[31] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *ICST '10*, pages 459–468. IEEE, 2010.

[32] K. Pohl, G. Böckle, and F. Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag New York Inc, 2005.

[33] H. Samih. Relating Variability Modeling and Model-Based Testing for Software Product Lines Testing. In C. Weise and B. Nielsen, editors, *Proceedings of the ICTSS 2012 Ph.D. Workshop*, pages 18–22, Aalborg, Denmark, 2012. Aalborg University, Department of Computer Science.

[34] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés. The Drupal Framework: A Case Study to Evaluate Variability Testing Techniques. In *VaMoS '14*, pages 11:1–11:8. ACM, 2013.

[35] S. E. Sprenkle, L. L. Pollock, and L. M. Simko. Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour. *STVR*, 23(6):439–464, 2013.

[36] J. Tretmans. Model based testing with labelled transition systems. In *Formal methods and testing*. Springer-Verlag, 2008.

[37] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2007.

[38] E. Uzuncaova, S. Khurshid, and D. Batory. Incremental Test Generation for Software Product Lines. *TSE*, 36(3):309–322, 2010.

[39] A. Van Lamsweerde. *Systematic Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2008.

[40] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.